ECSE 324 Project Report

Group Number: 55
Charbel El Hachem - 260667384
(Charles)Yin Jun Huang - 260743592

ECSE-324-001: Computer Organization

Professor Warren Gross

Thursday, December 07, 2017

Table of content

General Overview of our project:

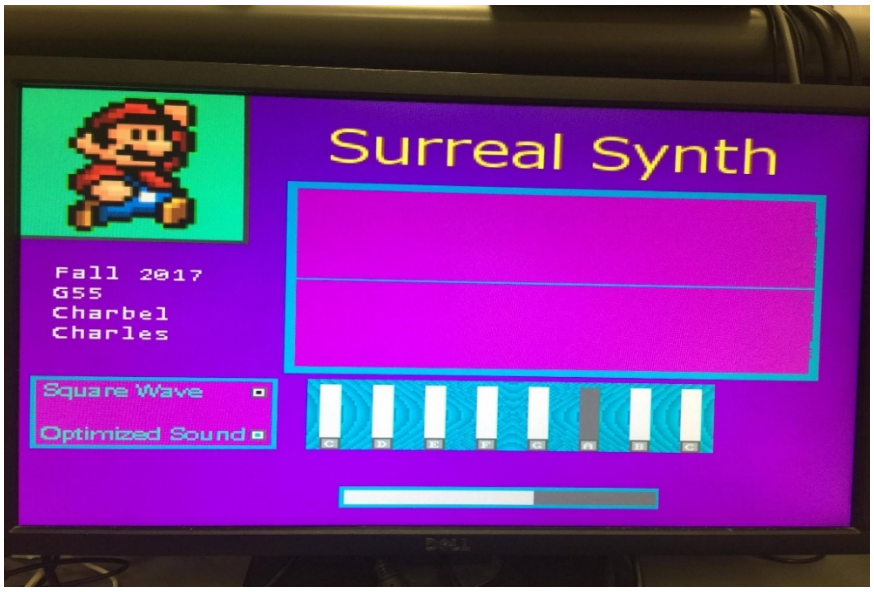

Figure 1: photo of the UI of our application

The goal of Lab 5 was to create our own sound synthesizer with a PS/2 keyboard input support and waveform drawing. Our project is named Unreal Synth. As the Figure 1 shows, our application has an advanced UI drawn using our image data parser. The middle part of our UI displays the different waves when different keys are pressed. Below the wave display section, we have our piano-style keys. The bottom part of every key indicates their notes. When any key is pressed, the color of the key will turn from white to grey to indicating that a note that is being played. At the bottom of our UI, we have a volume bar to indicate the current volume. At the left side of UI, we have our logo which is Mario, our group number, group members and additional features such as switching from sine wave to Square Wave and Optimized Sound mode which improves sound quality. When the Square Wave is selected, square waves are played instead of sine waves. The square waves will have same frequencies as sine waves for different keys. Moreover, since we are drawing the piano keys and dynamically displaying waveforms when we play a note, it may influence the audio quality hence the addition of the Optimized Sound feature which disables visual effects only allowing sound related code to be run increasing audio quality.

As a summary, our key features are :

- Unique user interface drawn through our image data parser
- Sound changing from sine wave to square wave(0 key)
- Sound optimization mode where visuals are disabled for best audio quality (9 key)
- Volume bar (+ and - keys)
- A dynamic waveform drawer
- Custom implementation of a double pixel buffer for waveform clearing
- Labeled piano keys display
- A Mario image drawn through our image data parser
- Responsive keyboard input and accurate soundwave addition

Sound wave creation and addition:

MakeWave( ):

Description:

This function returns a signal that is calculated at given frequency, sampling instant and amplitude.

Approach:

In this function, we use the given frequency, sampling instant and sampling frequency which is 48000 Hz to calculate the index of the signal in the sine wavetable allowing us to obtain the sine wave signal amplitude.If the enableSquareWave flag is set to 1, it instead returns a square wave signal value by having it return a high if the index is between 0 and 23999 and a low if the index is between 24000 and 47999.

MergeWave( ):

Description:

This function returns the merged signal value at a given sampling instant when one or multiple keys are pressed simultaneously.

Approach:

We have a global noteStatus array composed of key status flags and their corresponding frequencies as shown below.

```
double noteStatus[16]={0,0,0,0,0,0,0,0,130.813,146.832,164.814,174.614,195.998,220.000,246.942,261.626};/
```

The function will check for pressed keys by looping through each note status and checking its value, if the value is 1, it will then call makeWave with the corresponding key frequency and add the result to our signal value while keeping track of how many signals were added. Finally, we then divide the sum of signals by the number of added signals to normalize the amplitude.

Keyboard waveform manipulation:

UpdateKeyStatus( ):

Description:

This function goes through every possible key and updates the noteStatus array when any key is pressed or released. The function will return 1 if any key is being pressed. Otherwise, it will return 0.

Approach:

In this function, we have several if statements to check the data in the PS/2 data register. If the data matches one of the make codes of the notes, we will update noteStatus array by setting the flag of note to 1.If the data is 0xF0 which is the break code, we know one key is being released. Then, we check the make code after 0xF0, because the make code after 0xF0 indicates which key is being released. After we find which key is released, we update noteStatus array by setting the flag of the released key to 0. In

order to improve the overall run time of the function, we check note make codes before we check setting make codes and volume make codes since note make codes need to be processed very fast. Finally, we go through the noteStatus array and check if any key flag is 1, if so, the function return 1, otherwise, the function return 0.

Improvements:

We could use interrupts instead of polling for data. However we thought that having an ISR run at every single typematic rate would severely hinder the audio quality, which is why chose to opt out of interrupts. However, interrupts could speed up our software if the ISR implemented correctly.

Sound and waveform drawing:

PlaySound( ):

Description:

Main part of our software, it calls the updateKeyStatus() method to check for keyboard input and then outputs sound by calling mergeWave which will create a sample according to the pressed key, it also allows the continuous drawing of the sound wave.

Approach:

We start by initializing a timer with a timeout of 1000000/48000 which allows us to avoid surpassing the audio sampling frequency.

We then call updateKeyStatus function to check which buttons are being pressed and update flags. If buttons are being pressed, we then check if the button combination has changed from the last loop. The way to check is by multiplying the noteStatus values with prime numbers, and adding them into currentKeyValue in order to create unique value for all possible key combinations. This algorithm allows us to compare currentKeyValue with the previously computed keyValue. If the currentKeyValue is not equal to keyValue, it means the key combination has changed; we then call displayKeys() to display pressed keys.

We then create and store a signal using mergeWave(t), t is a global index variable which allows us to keep different waveforms being played in phase.

Afterward, since we obtained the signal amplitude at t, we will compute temp=t%6000 meaning that temp will be equal to 0 every 1/8th of a second since 6000/48000=1/8. It allows us to draw and update the waveform 8 times per second.

The waveform drawing is done by scaling down the resulting signal values, then drawing 480 consecutive samples while storing the drawn amplitudes in clearBuffer[480]. This buffer is our custom implementation of a double buffer; it is used when updating the wave form where the previously drawn pixels are replaced by our background color (which results in a pixel by pixel clearing) before drawing the updating waveform pixel. This allows us to only clear the pixels which need to be updated instead of clearing the whole rectangular area containing the waveform.

Finally, when the waveform drawing is done we write the signal to the output using audio_write_data_ASM().

Challenges:

We had to constantly find ways to optimize our calculations since this method is constantly looping and if delayed by computations will result and degraded sound quality, we had to find a waveform update rate which compromised between the visual and audio quality.

The wave form clear buffer was also implemented to speed up our waveform drawing process. We also found that using ternary operators to avoid computing a modulo when we switch to Optimized Audio mode, would take more time than simply computing the modulo, which was very interesting.

User Interface:

DisplayVolume( ):

Description:

Displays a volume bar which represents the current volume level.

Approach:

We obtained the currentAmplitude value, scaled it and using a for loop we filled the volume bar with white pixels when the currentAmplitude was higher than the amplitude represented by volume bar and filled the rest of the bar with grey pixels.

To optimize the drawing process, we unrolled the loop which draws the height of the volume bar since using loops takes more cycles than repeating the same instructions successively.

Improvements: Gradually change the color of the pixel when increasing the sound. However, this might cause clarity issues which is why we opted not to do that.

DisplayKeys( ):

Description:

Displays piano-style keys. When any key is pressed, the key color will change from white to grey.

Approach:

We are checking the noteStatus array from index 0 to 7 for pressed keys, and then we are accordingly drawing the piano keys with the appropriate color.

Challenges:

At first, we drew bigger sized piano-style keys, but that took too much time to draw it and it affected sound quality. Therefore, we chose to draw smaller keys.

ConvRGB( ):

Description:

This method converts html-style RGB values into a pixelBuffer short value.

Approach:

We scaled down the red value from 255 to 32, the green value from 255 to 64 and the blue value from 255 to 32, then using bit manipulations we shifted those values into place to form 16 bits short containing our desired RGB values which can be directly stored to the pixel buffer and therefore displayed through vga.

PixelEdit( ):

Description:

Replace a specific color in a raw image data to another one of our choice

Approach:

In our case it looks for a Green pixel (0,255,0) and replace it with our desired color, this is used to change mario's background color by directly editing the RGB values as they are being parsed from the raw image data.

Improvements:

We could add a threshold in order to also change very close color, such as slightly lighter green or darker green, the pixel value offset could then be applied to the color we are replacing them with in order to keep the contrast instead of replacing it with a flat color.

DrawUIBMP( ):

Description:

Parse given raw image data into pixelbuffer pixel format and draws them accordingly, the raw image data in this case is the ui interface

Approach:

We start by splitting the halfWord in the ui[115200] array(which is stored in bmp.s) into the con[230400] array which will contain all the r,g,b values of all the pixels that need to be draws, since the UI image is 320*240 we need to draw 76800 pixels which means we need to process 76800*3=230400 values.

Then we loop through rgb triplets and convert them into pixelBuffer format using our rgb converter, then we write the pixel and its corresponding color to the pixel buffer using VGA_draw_point_ASM(x%320,x/320,rgb), the modulo operator allows us to reset x after drawing one horizontal line of pixels and the divide operator allows us to increment y after drawing a horizontal line.

This loops until the entire array was processed resulting in the image drawing

Challenges:

We needed time to figure out how to export our image into raw image data, then spent time deconstructing the raw data format in order to understand the rgb data structure and therefore process them.

Improvements:

We could skip black pixel drawing since we are clearing the pixel buffer before calling this method therefore black pixel drawing is redundant.

Another thing we could have done, is try doing image data compression and decompression, since the size of bmp.s is around 1 megabyte it takes a while to compile it into the board, however this would take too much time to implement.

DrawBMP( )

Description:

Parses given raw image data into pixelbuffer pixel format and draws them accordingly, the raw image data in this case is a mario image

Approach:

It is the same approach as in DrawUIBMP, however since the size of the mario image is 80x80 we had to modify numbers to be able to draw it.

Improvements: Same as above.

DrawUI( ):

Description:

Draws the complete user interface

Approach:

We simply start by clearing the pixel and character buffers then call all the user interface methods, we also used VGA_write_char_ASM() to label the piano keys.